ABSTRACT

A way to measure the complexity of object-oriented software involves topological features of the code’s hierarchical organization at the method, class, package, and component levels. Sangwan et al (2008) suggested that as certain software products evolve, this complexity shifts from lower to higher structural levels, or vice-versa. They studied three widely used open source software programs and showed that these structural shifts called “epochs” were present and suspected that this phenomenon was pervasive. To support or refute this assertion, 30 open source programs were studied and structural shifts in complexity were found significantly in 27 of them. In those projects where no complexity shift was evident, no refactoring had occurred. These findings further suggest that in large, open source software projects, when refactoring occurs a shifting in complexity from one level to another will occur.

Keywords: Epochs, Open Source, Refactoring, Software Complexity, Structural Epochs

1. INTRODUCTION

Software complexity has been a subject of great interest since the work of Belady and Lehman (1976), McCabe (1976) and others. Belady and Lehman’s work led to “Lehman’s laws” of software evolution, which are largely informal observations about a software product’s tendency to become more complex, that is, more difficult to understand over time. McCabe’s cyclomatic complexity metric, which is a topological measure of code execution paths, or its close relative, weighted methods per class, is widely used in modern software metric tools such as SourceMonitor (2013) and in Structure 101 (2013).

More recently, Eick et al (2001) used multiple statistical methods and a rich data set to show that code decays over time. Informally they stipulate that code is “ decayed if it is more difficult than it should be.” In their study, code decay is more precisely reflected in the cost of the change, the time to complete the change and the quality of the changed software.

In a pioneering survey, Bandi, Williams & Allen (2013) examined different ways in which code decay has been historically evaluated. 30 peer-reviewed articles studying code decay over time and over successive versions of the code were reviewed. The authors found that decay and architectural degeneration was measured in two principal ways – through developer ob-
servations, and semi-automatically using code analysis tools and metrics.

Human-based assessments of code decay include manual inspections of source code and related historical project artifacts. In one example study, a web-based questionnaire was used to gather input from developers in detecting “code smells” such as bloated classes and long method parameter lists. Semi-automated assessments of code decay include the use of generated software metrics. Various metrics having to do with coupling were most indicative of source code decay. It was verified that violating architectural and design rules leads to code decay. Other aspects of the measurement of code decay were explored such as historical data analysis. This includes an evaluation of change-management data and the history of defect fixes. Additionally, quality attributes of source code were considered including maintainability, understandability and extensibility. There are also developer and process attributes that lead to code decay including novice developers, developers simply focused on functionality, the violation of object-oriented principles, and the adverse effects of tight deliverable deadlines.

Izurieta, C., & Bieman, J. M. (2007) studied code decay in relationship to design patterns in the evolution of the JRefactory program over a four-year period. The design patterns selected were Visitor, State, and Singleton. Modular and organizational code decay was considered. Modular code decay principally includes increases in coupling. Organizational decay includes decay in relation to source code files, object-oriented classes, and namespaces. Some modular decay was found with the Visitor and Singleton patterns. As patterns evolve they break down in modularity. The Adapter pattern alone showed increases in organizational code decay.

Capiluppi & Beecher (2009) focused on software structural components in relation to code decay. They studied 50 projects each from the Debian and SourceForge repositories. The Debian repository has a greater barrier to entry. Repositories that have higher barriers to accessibility, such as Debian, contain code that is more evolvable and maintainable.

Larger source code projects require extra active measures to reduce complexity and decay. The integrity of software decreases as the software evolves. Structural complexity grows as evolutionary activity increases. Debian projects have greater functional coupling due to interconnectedness. Not surprisingly, the authors found that Debian received greater complexity control work than did projects in SourceForge.

Dependencies between modules and packages have long been of interest but the identification of cyclic dependencies in object-oriented software is more recent. The literature advises developers to avoid such dependencies in order to improve quality attributes such as maintainability, comprehension, and testability (Melton and Tempero, 2007). McGregor (2006) suggests that complexity is what Brooks (1987) calls an essential aspect of software systems, not an accidental one, and that clear software architecture requires solid communication between developers and architects.

Dependency structure matrixes (DSMs), which were invented for optimizing product development processes can also be used to study the interdependency between software modules based on a standard notion of dependency where “a module A depends on a module B if there are explicit references in A to syntactic elements of B” (Sangal et al, 2005). A dependency structure matrix can be manipulated to establish whether circular dependencies exist. If so, they can be eliminated by partitioning, where offending modules are combined into a composite module. Sangwan et al (2010) applied DSMs to the open-source software project Hibernate.

Software’s structural complexity can permeate its hierarchical organization from individual methods through the class, package and component levels. Sangwan et al (2008) studied three open-source software projects, Findbugs, Hibernate and JFreeChart, and found that in these projects complexity tended to shift periodically between lower and higher structural levels so that the predominant structural complexity resides by turns at high and low levels. Excess complexity shifts from lower level constructs...
Strategic Capital Capacity and Capability Management of IS/IT
Teay Shawyun (2009). Strategic Information Technology and Portfolio Management (pp. 230-249).
www.igi-global.com/chapter/strategic-capital-capacity-capability-management/29747?camid=4v1a

Strategic Market and Customer Driven IS/IT Planning Model
www.igi-global.com/article/strategic-market-customer-driven-planning/52069?camid=4v1a